Spons & Shields: Practical Isolation for Trusted Execution

Vasily A. Sartakov
v.sartakov@imperial.ac.uk
Imperial College London
United Kingdom

Daniel O'Keeffe
daniel.okeeffe@rhul.ac.uk
Royal Holloway University of London
United Kingdom

David Eyers
dme@cs.otago.ac.nz
University of Otago
New Zealand

Lluís Vilanova
vilanova@imperial.ac.uk
Imperial College London
United Kingdom

Peter Pietzuch
prp@imperial.ac.uk
Imperial College London
United Kingdom

Abstract

Trusted execution environments (TEEs) give a cost-effective, "lift-and-shift" solution for deploying security-sensitive applications in untrusted clouds. For this, they must support rich, multi-component applications, risking a large trusted computing base inside the TEE. Fine-grained compartmentalisation can increase security through defense-in-depth, but current solutions either run all software components unprotected in the same TEE, lack efficient shared memory support, or isolate application processes using separate TEEs, impacting performance and compatibility.

We describe the *Spons & Shields framework* (SSF) for Intel SGX TEEs. Spons and Shields are new abstractions that generalise process, library and user/kernel isolation inside the TEE while allowing for efficient memory sharing. For unmodified multi-component applications in a TEE, SSF dynamically creates Spons (one per POSIX process or library) and Shields (to enforce a memory access policy). Applications can be hardened easily, e.g., by using a separate Shield to isolate an SSL library. SSF uses compiler instrumentation to protect Shield boundaries, exploiting MPX instructions if available. We evaluate SSF using a complex application service (NGINX, PHP interpreter and PostgreSQL) and show that its overhead is comparable to process isolation.

CCS Concepts: • **Security and privacy → Trusted computing**; **Operating systems security**.

Keywords: trusted execution, isolation, compartments, SGX

VEE '21, April 16, 2021, Virtual, USA

ACM ISBN 978-1-4503-8394-3/21/04…$15.00
https://doi.org/10.1145/3453933.3454024

ACM Reference Format:
Vasily A. Sartakov, Daniel O'Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. 2021. Spons & Shields: Practical Isolation for Trusted Execution. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21), April 16, 2021, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453933.3454024

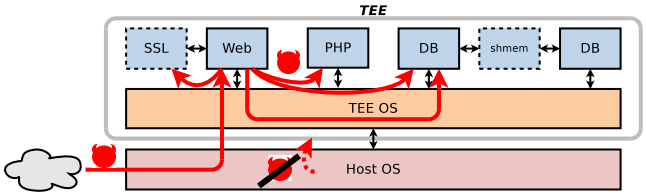
1 Introduction

Increasingly more cloud providers [2, 28, 48] offer hardware support for *trusted execution environments* (TEEs) [16, 26, 32]. TEEs enable cloud tenants to deploy applications that are protected from other privileged system software or direct access by cloud administrators. The initial use of TEE technologies [16, 26] followed an SDK model in which users developed security-sensitive software components running inside TEEs, resulting in a small trusted computing base (TCB).

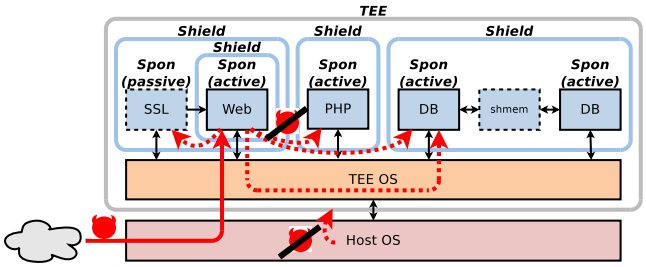
More recently, came *"lift-and-shift"* models, in which containerised applications or entire virtual machines (VMs) execute inside of TEEs. This requires a trusted OS inside the TEE (*TEE OS*). TEE technologies such as Intel SGX [16] support the execution of unmodified Linux applications by implementing library OSs as the TEE OS [4, 6, 21, 58]; AMD SEV-SNP [3] and Intel TDX [30] support full VM execution, which allows a regular guest OS kernel as the TEE OS.

The ease of "lift-and-shift" is popular with cloud tenants, but it raises security concerns due to the large TCB size of entire applications executing inside TEEs.

Traditionally, *compartmentalisation* [7, 59, 83] has been used as a defense-in-depth technique to contain vulnerabilities in large TCBs. It requires isolation mechanisms to separate software components from each other. We argue that existing isolation mechanisms, however, are insufficient to improve the security of large applications inside of TEEs: *Process isolation* is a coarse-grained isolation mechanism that assumes that an application is structured as a set of processes. Extracting security-sensitive functionality into separate processes, however, can be a complex task and subsequently impact application performance [62–64]. Process isolation



(a) Attacks without intra-TEE isolation



(b) Intra-TEE isolation provided by Spons and Shields

Figure 1. Isolation mechanisms for TEEs

also requires enforcement by the OS kernel and MMU hardware. Yet, OS and MMU support inside the TEE may be limited, as the TEE cannot rely on cooperation by the untrusted host. For example, SGX TEEs cannot implement fork-based process semantics faithfully and efficiently [5, 12, 72].

User/kernel isolation is designed to protect a trusted kernel from untrusted userspace applications. Inside TEEs, this threat model is inverted: typically an outside attacker uses the host interface to the TEE as an attack vector, e.g., launching Iago attacks [13] against the TEE OS. Traditional user/kernel space isolation therefore does not prevent vulnerabilities in application components from being exploited.

Multi-TEE isolation [9, 68] uses separate TEEs to isolate application components. For example, it has been used to implement process isolation for SGX TEEs [12]. This isolation approach, however, is inefficient: creating TEEs dynamically is expensive, and using multiple TEEs prevents efficient data sharing due to data copying and encryption overheads.

To understand the isolation requirements for TEEs, we consider the example in Fig. 1a, which shows a medical record service that is protected by a TEE. The service consists of four application components, which are not mutually isolated: (i) an NGNIX web server; (ii) an SSL library to encrypt external communication; (iii) a PHP scripting engine for business logic; and (iv) a PostgreSQL DBMS. For the service to execute, it must be able to create multiple processes inside the TEE and use inter-process communication (IPC) between them (Fig. 1a, top). For example, the web server, PHP and DBMS components execute as functionally separate processes, and PostgreSQL uses a pool of processes to implement functionality such as handling incoming requests.

From a security point-of-view, we require *process isolation* inside the TEE (Fig. 1a, top-right side) so that an attacker cannot use web server requests as an attack vector to access security-sensitive data in the memory of other processes such as the DBMS [65, 70]. We also require more fine-grained isolation between software components through *library isolation* (Fig. 1a, left). For example, by isolating the SSL library itself, we can protect its cryptographic keys from access after the web server has been compromised. Similarly, the PHP interpreter should be isolated from its modules, which are written in type-unsafe C code. Finally, we want to protect the TEE OS through traditional *user/kernel isolation*: without it, a compromised web server may tamper with the TEE OS (e.g., to access other processes), thus bypassing process isolation. Processes should only be able to access the TEE OS and security-critical libraries via controlled entry points.

Despite the above need for fine-grained isolation, isolated components must also efficiently *share memory* for performance reasons. For example, PostgresSQL processes use shared memory IPC to exchange data; the SSL library must access memory of the web server to implement its plug-in model. Process memory must also remain accessible by the TEE OS, e.g., to control user-level thread scheduling.

We describe the **Spons & Shields framework (SSF)**, which introduces two abstractions, **Spons** and **Shields** (see Fig. 1b). The two abstractions provide a uniform mechanism for fine-grained compartmentalisation in TEEs, unifying process, library and user/kernel isolation while permitting efficient memory sharing. By disentangling the execution and memory protection aspects of isolation, SSF can provide efficient mechanisms that fulfil the functional and security requirements for efficient and practical intra-TEE isolation:

(i) Spons encapsulate the *execution contexts* of application components, as well as the entry points exported to other Spons. Developers can create Spons dynamically at low cost to express multi-programming and concurrency. Multi-threaded processes are supported by **active Spons**, which contain one or more execution contexts controlled by the user; **passive Spons** can shield libraries as portions of an active Spon. Spons can call into other Spons only via well-defined entry points, and passive Spons obtain an execution context from their caller (either a passive or active Spon).

(ii) Shields define a hierarchy of *memory protection* boundaries across Spons. This hierarchical approach allows developers to use **nested Shields** to support the isolation shown in Fig. 1b: (1) the TEE OS sits at the root, with system calls as well-defined entry points, to provide user/kernel isolation; (2) the web server, PHP and DBMS processes are defined as nested Shields to provide process isolation. All DBMS processes use the same Shield to permit shared memory IPC; and (3) the SSL library is nested between the OS and the web server, which offers library isolation to protect the cryptographic keys, while allowing the library to access the web server's connection data with minor application changes.

Interestingly, this hierarchical approach of Shields allows for *efficient* and *practical* memory protection without assuming virtual memory and OS support by the TEE hardware. Given the limited hardware support in some TEEs, a TEE OS

may provide process functionality but no isolation among them [6, 34, 37, 45]; others provide process isolation at the expense of performance [12] and support for shared memory [12, 72] (e.g., a requirement for PostgreSQL).

Our SSF prototype uses Intel SGX because it is a minimalist extreme in TEE design. SGX restricts hardware features the TEE OS can use (e.g., the MMU can only be used with cooperation by the untrusted host OS). Since SGX provides no hardware mechanisms to enforce intra-TEE isolation, SSF uses compiler-based instrumentation—an LLVM pass [39]—to protect Shield boundaries using well-known *software fault isolation* (SFI) [81] and *control flow integrity* (CFI) [1] techniques. To reduce the overheads, SSF exploits MPX instructions to perform bounds checks [61].

The Linux kernel library (LKL) is our SSF implementation's TEE OS, which provides a full Linux ABI inside SGX TEEs. It utilises user-level scheduling for security [84] and performance reasons [4, 53]. To maintain user/kernel isolation, SSF therefore partitions the standard C library, dividing threading metadata between the TEE OS and the application.

We evaluate SSF with two applications in SGX TEEs, a complex service similar to the healthcare scenario above (NGINX, PHP, and PostgreSQL) and the PyCryptodome cryptographic framework [54]. Our results show that PostgreSQL and PHP only see overheads of 10% and 22%, respectively, compared to a non-isolated system; NGINX can isolate its SSL library with negligible overheads; and Python isolates the PyCryptodome C extensions with 21%–57% overhead. Compared to a multi-TEE design, SSF spawns processes 7× more quickly and handles requests with 27% lower latency.

2 Isolation Support for Trusted Execution

Next we provide background on hardware TEEs (§2.1) and the OS support within TEEs (§2.2). We then discuss the requirements for intra-TEE isolation (§2.3) and survey existing isolation approaches (§2.4).

2.1 Hardware TEEs

Data processing in cloud infrastructures entails a major risk: the cloud provider has privileged data and code access. Major CPU vendors responded with extensions to support *trusted execution environments* (TEEs). ARM CPUs provide *TrustZone* [26], which isolates a single "secure world" TEE from the rest of the CPU, limiting its applicability in multi-tenant IaaS scenarios. AMD offers *Secure Encrypted Virtualization* (SEV) [32], targeting runtime encryption for VMs.

In contrast, Intel's *Software Guard Extensions* (SGX) [47] provide a minimal TEE targeted at application-level code, where SGX enclaves (TEE instances) lack typical hardware mechanisms to manage privilege levels and page tables. SGX enclaves use memory that is isolated from the rest of a process' address space. The physical memory of enclaves, as part of the *Enclave Page Cache* (EPC), is located inside the *Processor Reserved Memory* (PRM) region. The hardware controls access to the PRM and transparently encrypts its contents using the *Memory Encryption Engine* (MEE), thus protecting enclaves from adversaries with physical access [23]. SGX enclaves require a mix of user-space OS support (using a TEE OS linked against the application) and interaction with the host OS – which is outside the enclave's TCB – to perform operations such as I/O or dynamic EPC page allocation [31, 46].

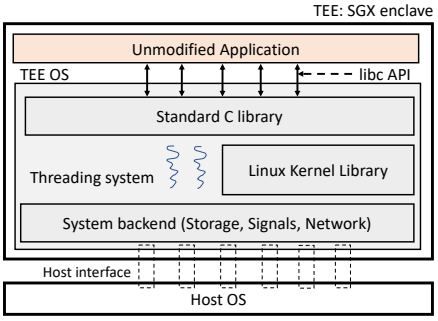


Figure 2. TEE OS architecture using library OS approach

2.2 OS Support Within TEEs

TEEs can be used to protect complex POSIX applications from an untrusted host. TEEs such as AMD SEV allow executing an unmodified, virtualised guest OS (as the TEE OS). While this offers an easy "lift-and-shift" approach, it also results in a large TCB by running a large unmodified stock OS kernel. In contrast, TEEs such as Intel SGX require a bespoke *library OS* as the TEE OS that offers OS functions to applications. Existing library OSs for TEEs provide various levels of OS functionality, ranging from memory allocation, file and network I/O, to thread scheduling and synchronisation [4, 6, 12, 58]. By tailoring the TEE OS to the TEE, large parts of traditional OS kernel code, e.g., related to device drivers, low-level hardware management and multi-user support, become unnecessary, leading to smaller TCB sizes and stronger security. Therefore we assume a library OS based approach for the TEE OS in the rest of the paper.

Fig. 2 shows a representative design of an application deployed with a library OS as the TEE OS. The application is linked against a *standard C library* interface, which is exposed by a TEE OS layer. The TEE OS layer includes the implementation of the standard C library (e.g., *musl libc* [50]), which performs direct function calls to the system call implementations in the TEE OS instead of using hardware traps. The TEE OS provides the OS functionality required and may be implemented from scratch [12] or may reuse parts of an existing OS kernel (e.g., the *Linux Kernel Library* (LKL) [60]).

2.3 Threat Model and System Requirements

We consider a threat model in which the application code may contain vulnerabilities. The TEEs do not trust the host OS, which can be malicious or compromised, nor external clients interacting with the TEE. We assume that the TEE hardware implementation is correct and do not consider side-channel attacks, which can be addressed orthogonally using

Table 1. Isolation approaches for trusted execution (✗/✓ indicates requirement partially satisfied).

Approach		Description	(R1) Fine-grained isolation	(R2) Efficient memory sharing	(R3) POSIX compatibility	(R4) TEE implementable
Domains	ERIM [78] Shred [14] CubicleOS [67]	Process-level domains with isolated threads	✗/✓	✗	✓	✓
ISA extension	IMIX [18] CHERI [83]	Hardware isolation primitives	✓	✓	✓	✗
Kernel-level	LwCs [42] SeCage [43]	Kernel/Hypervisor/MMU enforced isolation	✗/✓	✓	✓	✗
Compiler	SGXBounds [38] ConfLLVM [8]	Code instrumentation	✗/✓	✓	✗	✓
	Occlum[72]	Process isolation for TEEs	✗	✗	✗/✓	✗/✓
	SSF	Decoupled TEE execution/isolation abstractions	✓	✓	✓	✓

well-known techniques [35, 41, 79, 86]. We also assume that the TEE OS and the compiler are implemented correctly, and thus part of our runtime TCB. Our goal is to protect untrusted application components (e.g., libraries or processes) from each other, and also protect the TEE OS from them.

Intra-TEE isolation approaches must satisfy requirements:

(R1) Fine-grained isolation. The approach should provide primitives to compartmentalise the components of TEE applications [22]. Compartmentalisation should be applicable to both processes and libraries with little developer intervention and low performance overhead. In addition, the library OS inside the TEE should be regarded as another (set of) software components, thus unifying user/kernel isolation.

(R2) Efficient memory sharing. Inter-component communication is performance critical for many applications. The approach should support efficient shared memory communication between a subset of application components when required, while isolating components from the rest of the application and the TEE OS.

(R3) Compatible with existing POSIX applications. The approach should provide abstractions that are compatible with existing applications built from processes, threads and types of IPC. The primitives should be available at runtime and not impose restrictions on the number of execution and isolation units (e.g., threads, processes and compartments), only subject to the available memory.

(R4) Implementable within TEEs. Compartmentalisation should be available inside TEEs and be compatible with their security model. In particular, the soundness of compartments should not depend on support from the untrusted host or access to hardware features not widely available on TEEs.

2.4 Existing Isolation Approaches

Since a single process can host more than one TEE, a strawman solution for isolation is to spawn multiple TEEs [12, 68, 73]. A multi-TEE design, however, must use encrypted communication between TEEs. Such communication may require partial redevelopment of an application (e.g., to move from RPCs to message-passing), and the required encryption can slow down data exchange by up to 10× [68]. A multi-TEE design thus does not satisfy **R2** or **R3** due to the extra development effort and performance costs.

In Tab. 1, we compare other approaches for *intra-process* isolation of userspace code (see §7 for more related work).

Intra-process domains. Different hardware mechanisms can be used to compartmentalise processes. Intel MPK [29] enforces domains inside a process by assigning tags to memory pages. Similarly, tags can be assigned to threads, binding them within particular domains. Shred [14] (uses Arm domains that are similar to Intel MPK), ERIM [78] and CubicleOS [67] introduce system abstractions for MPK, but only a few (16) isolated contexts can be used, contradicting **R3**. libMPK [55] lifts this limitation by virtualising protection keys but requires trusting the OS kernel, contradicting **R4**.

Hardware isolation extensions. Researchers have also proposed new hardware extensions for isolation. IMIX [18] introduces in-memory isolation for x86 that allow developers to mark memory pages as *security sensitive*. CHERI [83] and CODOMs [80] introduce hardware-supported capability systems, which support program compartmentalisation. All these approaches rely on hardware extensions unavailable on commodity TEE platforms, contradicting **R4**.

Kernel-enforced isolation. The OS kernel can isolate parts of a process using its own primitives and the MMU. Lightweight Contexts (LwCs) [42] are an abstraction for intra-process isolation. Each LwC has its own heap and stack but can access only limited memory ranges. Switching between LwCs involves the OS kernel, as it changes virtual memory mappings, file table entries and more. As with other proposals such as SeCage [43] or Secure Memory Views [27], for some TEEs, this requires relying on a trusted host OS/hypervisor, violating **R4**.

Compiler instrumentation. TEE code can be instrumented by the compiler to protect pointers. Using Intel MPX [61] or SGXBounds [38], developers add checks to all pointer-related

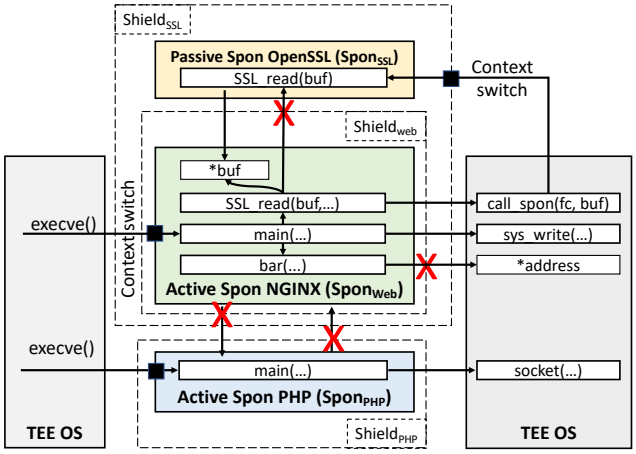


Figure 3. Anatomy of Spons and Shields

operations. MPX uses hardware bounds registers to check buffers; SGXBounds encodes buffer sizes into unused bits of SGX TEE pointers. Both can protect SGX TEE code but do not offer a programming abstraction for existing multi-process applications, contradicting **R3**.

Compiler instrumentation can also protect code at a coarser granularity. ConfLLVM [8] creates two partitions inside a process, one trusted and one untrusted. It instruments the untrusted partition and guarantees that its code can never reach the trusted one. It only supports two partitions, contradicting **R3**. Occlum [72] supports multi-process isolation inside SGX TEEs using MPX. However, it does not support library isolation (**R1**) and only supports a very limited form of memory sharing between pairs of processes with overlapping memory ranges (**R2**). Occlum also relies on the host OS for scheduling and synchronisation, reducing performance through costly TEE transitions and making it possible to introduce and exploit races in otherwise safe code (**R4**).

In summary, while many techniques have been proposed to isolate userspace components, they either require additional hardware, the involvement of the untrusted host OS kernel, or do not offer a suitable programming abstraction.

3 Spons and Shields

This section introduces *Spons* and *Shields*, the core abstractions in SSF to isolate execution in SGX TEEs. We give an overview of Spons and Shields (§3.1), describe their usage (§3.2) and explain their API (§3.3).

3.1 Overview

Fig. 3 shows a TEE with multiple Spons and Shields to protect the NGINX web server and the OpenSSL library in our example healthcare scenario. **Spons** encapsulate execution contexts: executable code with a set of known function entry points, per-thread contexts and stacks, and a heap allocator. SSF supports two types of Spons: **passive Spons** (*Spon$_{SSL}$*) encapsulate arbitrary code components with entry point functions specified by the developer (`SSL_read` in the example); **active Spons** (e.g., *Spon$_{web}$* in Fig. 3) are used to encapsulate POSIX processes (with their `main` entry point).

All user code is associated with a Spon, which serves both as an execution context handle and as the minimum unit of memory protection. Memory protection policies are expressed in SSF by assigning one **Shield** to each Spon (multiple Spons can be assigned to the same Shield), and by defining a hierarchical *nested* relationship between Shields. SSF then enforces the following invariant: Spon S is allowed to access all the code and data of all Spons contained in the Shield immediately enclosing S (recall that a Spon encapsulates the code, data, and heap associated with it).

The TEE OS executes on a default, outermost Shield that is also assigned by default to all Spons that did not specify a Shield. To make threading more efficient and secure against certain attacks from the host, each Spon has a set of user-level threads multiplexed over host OS threads [4, 53, 84]. Invoking a function of another Spon through the TEE OS results in a user-level context switch (see §4). Note that passive Spons are called synchronously, but contain independent thread stacks to preserve isolation from their calling Spons.

SSF instruments Spon code to check every memory access lies within its assigned Shield. Spons cannot directly access the TEE OS; instead, SSF deploys a callback table on each Spon with pointers to trusted trampolines into the TEE OS. Calls into the TEE OS thus have no context-switching overhead, and the trusted TEE OS itself is not instrumented. Note that the TEE OS is hardened to prevent unauthorised cross-Shield interactions (see §4.3), and our threat model relies on users deploying instrumented application code (see §2.3).

Users can declare Spons and Shields when starting a TEE (by mapping program paths to Spons) to compartmentalise unmodified applications (see §3.3). The same operations are also available to dynamically create Spons and Shields inside a TEE, allowing more sophisticated use cases (see §3.2).

3.2 Use Cases for Spon and Shield Separation

The simplest use case for SSF is to have one Shield per Spon – a *symmetric configuration*, which achieves the equivalent of conventional process-based isolation. This configuration works between processes of the same program e.g., processes isolating network connection handling [51, 76], or between the web server and PHP processes in our healthcare scenario.

Conversely, SSF supports multiple processes sharing memory in the form of "one Shield for multiple Spons". Shared memory is not readily supported by existing single-TEE [72] or multi-TEE solutions [12] when isolating processes from each other. For PostgreSQL with SSF (see §6.1), each DBMS process is placed on its own Spon but assigned to a single Shield, allowing access to shared memory. This policy is weaker than process-based isolation (all DBMS processes can access each other, not just the shared memory region), but

Table 2. SSF interface to manage Spons and Shields

API function	Description
Ahead-of-time and run-time operations	
`sid_t` **`alloc_shield`**`(sid_t sid, size_t size)`	Allocate `size` bytes for a Shield (nested if `sid` ! = 0)
`int` **`free_shield`**`(sid_t sid)`	Free an empty Shield `sid`
`int` **`register_spon`**`(sid_t sid, const char *path, size_t spon_size)`	Once created, allocate a Spon from `path` with Shield `sid` and `spon_size` bytes
`int` **`deregister_spon`**`(sid_t sid, const char *path)`	Remove record about `path` from Shield `sid`
`func_type` **`call_spon`**`(const char *path, const char *func_name, ...)`	Create a pre-registered passive Spon and call `func_ptr` from it (arguments must be scalar, or within the inner Shield)
Run-time operations	
`void *`**`alloc_mem`**`(sid_t sid, size_t size)`	Allocate `size` bytes from Shield `sid`
`int` **`free_mem`**`(void *ptr)`	Return buffer `ptr` to the Shield
Mapping of POSIX-like to SSF operations	
`int` **`execve`**`(const char *path, char **argv, char **envp)`	Create and run a Spon from a pre-registered `path`
`int` **`execves`**`(sid_t sid, const char *path, char **argv, char **envp)`	Create and run a Spon from `path` inside `sid`

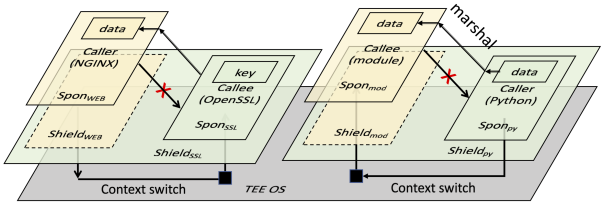


Figure 4. Asymmetric memory protection with Shields

it does not require application changes to provide security guarantees across other application components.

Fig. 4 shows a configuration not available in conventional processes (see §6.1 for details). The separation of Spons and Shields and the nesting of Shields permits *asymmetric memory protection*, which is crucial for the defense-in-depth protection for non-trivial applications.

In the left-hand figure, the hardened version of the OpenSSL library uses a passive Spon to isolate its private encryption key (variable `key`) from the rest of the web server. $Spon_{SSL}$ is a passive Spon that can access all memory inside $Shield_{SSL}$, which includes $Spon_{web}$. This way, OpenSSL can directly access the network data received by $Spon_{web}$ (variable `data`). Conversely, $Spon_{web}$ is an active Spon that can only access its own memory (i.e., it is constrained by $Shield_{web}$), and must use the TEE OS to access the decryption function in $Spon_{SSL}$ (i.e., the entry point `SSL_read` registered by $Spon_{SSL}$).

In the right-hand figure, a Python interpreter (active $Spon_{py}$) calls an unsafe C module (passive $Spon_{mod}$); this is useful to isolate native libraries such as pyOpenssl and Numpy that may have vulnerabilities. The module is constrained by creating the nested $Shield_{mod}$. In this case, the caller ($Spon_{py}$) marshals the function arguments into the callee ($Spon_{mod}$) and calls into an exported function through the TEE OS, which performs a context switch into the callee's Spon. When $Spon_{mod}$ executes, it is constrained to its Shield, hardening the application against attacks to the memory-unsafe C module.

3.3 Spon API

SSF extends the TEE OS POSIX primitives with additional calls to manage Spons and Shields. Tab. 2 shows the API calls used to configure an arbitrary number of Spons and Shields:

(1) Ahead-of-time operations. The first group of calls creates a set of rules in the TEE OS to configure an unmodified application when creating a TEE. The `alloc_shield` and `free_shield` operations manage the size and hierarchy of Shields. The `sid` argument in `alloc_shield` allows the creation of nested Shields (see §3.2), whereas the `size` argument establishes the maximum aggregate memory size that all its associated Spons can allocate, setting the per-Spon heap space. The calls `register/deregister_spon` modify the set of rules in the TEE OS that map executable paths to Spons and assigned Shields.

Calling `call_spon` creates a passive Spon for a `path` previously registered with `register_spon`. The TEE OS allocates space, loads the `path` into the target Spon's Shield, and dynamically creates a code thunk in the caller's Spon that application code uses to call into `func_name`. The `call_spon` operation only accepts public symbols exported by `path` (similar to dynamic linking), and the generated code thunk directs execution into the TEE OS to context-switch into the user-level thread of the target passive Spon. Any argument passed to this code thunk must be a scalar, or a pointer to memory allocated within the target Spon's Shield (see next).

(2) Run-time operations. The `alloc/free_mem` calls manage a Shield's memory. The TEE OS uses these calls to perform per-Shield allocations, e.g., when loading a program. Applications can also use them to have shared memory, e.g., to allow the Python interpreter to allocate the `data` buffer of the $Spon_{mod}$ in Fig. 4, which is then used to pass the arguments of `call_spon`'s resulting thunk.

(3) POSIX-like operations. SSF provides POSIX-like `execve` and `execves` calls. They consider the binary code paths registered with `register_spon` to instantiate the necessary Spons and Shields. Other functions such as `posix_spawn` and `exec*` can easily be supported using the same approach.

On a `path` match, SSF uses the assigned Shield memory to store the Spon's code, heap and stack. SSF also maps memory allocation primitives to `alloc/free_mem` using the calling thread's Shield. This is similar to how `mmap/munmap` are used to implement user-level sub-allocators.

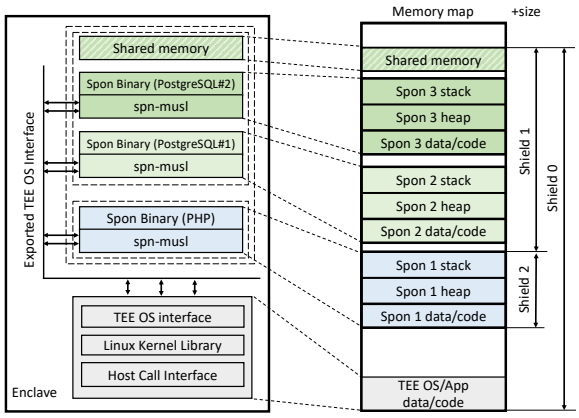


Figure 5. Memory layout of an application with SSF

Example of ahead-of-time configuration: Let us consider how to deploy the healthcare application scenario without code changes: (1) use `alloc_shield` to create three, non-nested shields, $Shield_{web}$, $Shield_{PHP}$, and $Shield_{DB}$; (2) register the Spons for all components with `register_spon`, assigning each to their own Shield; and (3) when the application starts, each call to `execve` triggers the creation of the necessary active Spons and Shields, and invokes their `main` function.

4 SSF Implementation

Next we explain how Spons and Shields are implemented in SSF. We describe the memory layout (§4.1), how memory accesses are constrained (§4.2), how the TEE OS interface is protected (§4.3), how to support multi-threaded execution (§4.4) and how to deploy applications with Spons (§4.5).

4.1 Memory Layout

In SSF, applications inside an SGX TEE consist of three parts (see §2.2): (i) the deployed application, (ii) a shared library with a standard C library interface (`musl` [50]), and (iii) the TEE OS kernel (LKL [60]). A TEE OS layer combines these components and provides SGX-related functionality, such as a host interface, user-level threading and locking.

The SGX TEE starts executing an `init` program that is linked against all the TEE OS components, and is contained in the outermost "zero" Shield, which has access to all the TEE memory (never shown for brevity). Fig. 5 shows the memory layout of the `init` program and the Spons and Shields that it creates by sub-dividing the available TEE memory. Each Spon has its own `text`, `data`, and `bss` segments loaded into the memory of its respective Shield, as well as an independent dynamic memory sub-allocator and per-thread stack.

To execute programs inside a Spon, we statically link them with `spn-musl`, a libC-compatible library for Spons. SSF then instantiates a *callback table* for each Spon to redirect invocations to required TEE OS functions.

4.2 Execution and Memory Access Isolation

SSF must enforce Shield boundaries without relying on the untrusted host OS (**R4**). Since Spons and Shields can be created dynamically, we must not limit the maximum number of isolated regions (**R3**).

SGX does not offer hardware support for memory protection. SSF thus combines *software fault isolation* (SFI) [81], *control flow integrity* (CFI) [1, 77], and MPX hardware acceleration [52, 61] to perform bounds checks on memory accesses. The combination of SFI and CFI is aimed at thwarting external attacks that try to bypass the isolation guarantees of SSF [65, 70], and supports unmodified application code.

To protect Shield boundaries, SSF "flattens" their hierarchical relationship, as defined through `alloc_shield`, into memory ranges in the TEE's virtual address space. Every non-nested Shield simply gets consecutive memory ranges of a known size, specified by argument `size` in `alloc_shield`, whereas each nested Shield recursively gets a portion of the memory range assigned to its immediately enclosing Shield.

Note that users are allowed to deploy Spons in the "zero" Shield, in which case no instrumentation is necessary for that code (i.e., it exists in the same Shield as the TEE OS).

Compiler-based bounds-checking. SSF provides a new compiler pass (*SSFPass*) that add bounds-checking instructions to every memory access in a Spon; code accessing memory outside the Shield will generate an exception.

SSFPass reserves one MPX bounds register (`BND0`) that contains the bounds of the currently active Shield, and inserts the necessary upper and lower bounds check instructions (`bndcl` and `bndcu`) for every memory access in the Spon. The pass also verifies that instrumented code does not modify the bounds register, which is only updated by the TEE OS when switching between Shields (via `bndmk`).

SSFPass is implemented using LLVM version 9.0 [39], with roughly 300 lines of C++ code. The pass operates on the intermediate representation (IR) of a Spon program linked with `spn-musl` (see §4.5) and does not require source code changes. It also supports assembly code if implemented as C inlines, for which it adds argument protection. Note that MPX support cannot be disabled by a malicious host OS because it is controlled by the untrusted `XCR0` register (managed by the TEE), and SGX ensures its integrity.

To decrease instrumentation overhead, SSFPass elides bounds checks on addresses known to be safe; e.g., temporary stack variables or standard memory-related functions with a known buffer size (such as `memcopy`, `memset`, or `memcmp`), which can be checked just once during their first access. Further compiler optimisations that remove unnecessary instrumentation could be added to SSFPass [72, 87].

Enforcing control flow integrity. SSFPass leverages LLVM's fine-grained forward-edge CFI [77] to restrict indirect function calls. It enforces that function calls take place using a function of the correct dynamic type, matching the static type originating from the call [44].

This approach provides limited CFI guarantees with "forward-edge protection" (i.e., calls), but not "backward-edge

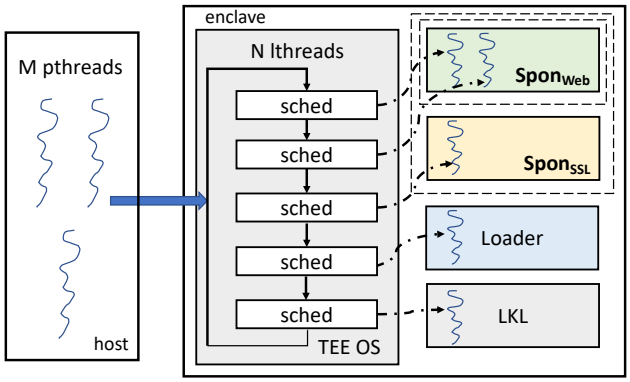


Figure 6. User-level multi-threading with Spons (Spon threads are scheduled by the TEE OS lthread scheduler.)

protection" (i.e., returns). More extensive techniques could be readily applied [11, 49], and we anticipate future hardware enhancements such as Intel's recently announced CET [71] to provide full protection with low performance overheads.

4.3 Protecting the TEE OS Interface

The system call interface and user/kernel mode separation protect the host OS. SSF's TEE OS is the in-enclave equivalent of the host OS kernel, and it must thus provide similar isolation guarantees when Spons interact with it.

Consider the following two functions: `memcpy(void *dst, const *src, size_t n)` to copy memory, and `write(int fd, void *buf, size_t count)` to write to a file descriptor. In both cases, an attacker may use a valid `src` pointer with a large `size` value that reaches outside the caller's Shield.

The `memcpy` function is implemented by `spn-musl` and therefore secured by the caller's memory protection (see §4.2). The `write` function, however, is implemented by the TEE OS and invoked via the callback table (see §4.1). As in other OS kernels, all public TEE OS functions check that all arguments are bounded by the calling Shield's memory region.

The functions exported to a Spon's callback table are defined at compile time, and can be filtered through a white list. The TEE OS interface supports the following functions: (i) 27 functions cannot be performed or emulated inside LKL and are mapped onto the host interface (e.g., `SYS_write` to an I/O device outside the enclave); (ii) 12 SGX-LKL functions are emulated within the enclave (e.g., `SYS_futex` and all `pthread` functions are implemented by the user-level threading library);[1] and (iii) LKL implements over 300 system calls, of which 119 use pointer arguments.

SSF ensures the validity of pointer arguments for all functions of the first and second types (39 in total), as well as for 30 functions from the system call interface that are used in the evaluation. Other functions can be similarly protected.

[1]This group of functions also includes those that bridge LKL and `musl`, as well as some debug functions.

4.4 Multi-Threading Support

SSF's TEE OS uses user-level threading (see §2.2). Fig. 6 shows the intra-TEE threading model used by the TEE OS (SGX-LKL). Each hardware thread (i.e., a *pthread* in the host OS) has a user-level scheduler that manages a pool of user-level *lthreads*. The user-level scheduler simply dequeues a runnable lthread descriptor from the pool, and context-switches to the target lthread.

The default implementation in SGX-LKL assumes no intra-TEE memory protection. This is incompatible with SSF because certain data (e.g., the thread stack and the `errno` variable) must be accessed by both the Spons and the scheduler. Solving this is challenging, since moving the scheduling logic into the Spons would require using the host OS during enclave run-time to map each lthread into a pthread and using its synchronisation primitives. Using the host OS not only incurs additional overheads, but also allows it to introduce and exploit data races in the TEE code [84].

To solve this problem, SSF maintains a centralised user-level scheduler in the TEE OS and instead *partitions* the lthread-specific data: lthread-related objects that are not used by Spons are allocated inside the TEE OS, while the rest are explicitly allocated within a Spon's Shield. We also remove thread-related code from `spn-musl`, redirecting calls to the TEE OS. This guarantees that all threads and synchronisation primitives use the same functionality, and Spons do not refer to thread metadata outside of their Shield.

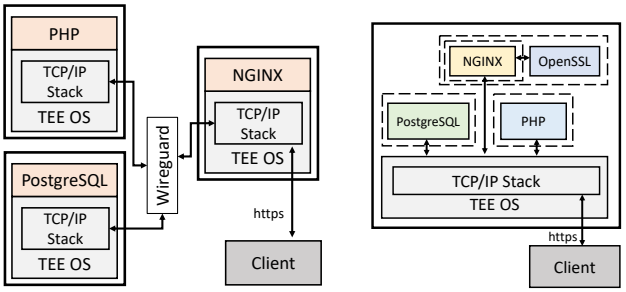
4.5 Building and Deploying Programs With SSF

Compilation of SSF programs is not dissimilar to ordinary TEE programs (**R3**). Programs must be compiled to an ELF file or a shared library. The SGX-LKL TEE OS currently only supports the `musl` standard C library [50], so application code must be linked against it.

The compilation process works as follows: (i) the program's source code is compiled into its LLVM intermediate representation (IR); (ii) the IR code is linked with the IR of `spn-musl`; (iii) the linked code is then processed by SSFPass and LLVM's CFI pass;[2] and (iv) the resulting code is compiled into a TEE PIE library.

SSF has an application *loader* that works as follows: (i) allocates a memory region for the target Spon using its assigned Shield; (ii) loads the Spon binary into this region and populates the callback table to link it with the shared TEE OS interface; (iii) allocates and populates per-Spon variables such as `environ`, `errno`, `argc`, `argv` and `envp`; (iv) allocates memory for the stack of the first thread (the remaining memory is used for the heap); and (v) sets up the MPX memory bounds register for the Spon's Shield and creates its first thread.

[2]The CFI pass assigns types to the functions in LLVM's IR and builds a jump table, which is used to validate dynamically matching function types in forward-edge indirect control flow transfers.



(a) Multi-enclave deployment **(b)** Deployment with SSF

Figure 7. Enclaved, multi-component web service

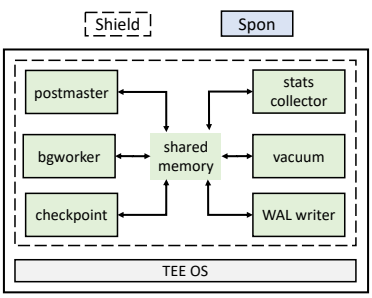


Figure 8. PostgreSQL with Spons

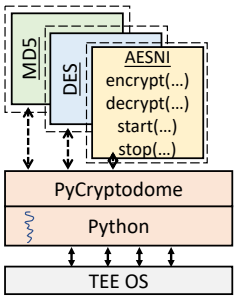


Figure 9. Module isolation

SSF does not preclude the use of security-sensitive instructions such as `EMODPE` [72]. Existing techniques are applicable here, e.g., an additional code generation pass to avoid inserting such instructions [78], or compiler labelling with post-compilation binary inspection [72].

5 Discussion

SSF offers better, yet portable compartmentalisation abstractions for TEEs. It uses TEE technologies (SGX here) to remove trust from the host OS, enforces the isolation between application components as well as the trusted TEE OS, and provides a narrow host interface to minimise reliance on critical host OS features (fulfilling our intended threat model).

Spons and Shields can be applied at arbitrary granularities, from processes to libraries (R1 in §2.3), support shared memory by using a single Shield for multiple Spons (R2) and are thus able to enhance the security of existing POSIX applications (R3). SSF also offers insights into the minimum requirements for compartmentalisation inside a TEE (R4). SGX is particularly challenging due to its lack of extensive hardware support, and we show a promising direction that couples simple hardware support (MPX) with software mechanisms (SSFPass and the TEE OS primitives).

More importantly, the semantics of Spons and Shields allow SSF to take advantage of both existing and future hardware mechanisms to further improve compartmentalisation performance. For example, we envision future incarnations of SSF using hardware such as CHERI capabilities, Intel MPK, or Intel's CET extensions. Finally, note that SSF uses SGXv1 and thus cannot self-manage execution permissions on pages; nevertheless, switching to SGXv2 would close that gap by using the new `EMODPE` and `EMODPR` instructions.

6 Evaluation

We explore the performance of SSF using two real-world applications: a multi-process multi-component web service, as introduced in Fig. 1 (§6.2) and a Python cryptographic library (§6.3). We also measure the overhead of the compiler instrumentation (§6.4), and the cost of Spon creation (§6.5).

We deploy the workloads on SGX-enabled servers with Intel Xeon E3-1280 v6 CPUs (microcode version `0xca`), each with 4 cores at 3.90 GHz, 8 MB of LLC and disabled hyperthreading and Turbo Boost. The servers have 64 GB of RAM, a 10 Gbps NIC, and run Ubuntu Linux 18.04 with Linux kernel 4.15.0-46. The version of the Intel SGX driver is 2.5.

6.1 Application Use Cases

We consider two application use cases for fine-grained compartmentalisation: (i) a web service inspired by our healthcare example with a mix of active/passive Spons and nested Shields; and (ii) a Python cryptographic framework that uses passive Spons for hardening against vulnerabilities in its C libraries. These represent different types of workloads: the former involves a pointer-heavy interpreter, an I/O-intensive web server, and a database with a complex child/parent lifecycle; the latter heavily invokes isolated modules.

Healthcare web service. We evaluate a typical LAMP-based architecture, with a front-end web server (NGINX v.1.16.1), a PHP interpreter (v.7.3.7), and a database backend (PostgreSQL v.12.2). The NGINX web server plays a crucial role in terms of security – it establishes encrypted network connections with its clients, and redirects their requests to the PHP interpreter, which in turn, interacts with the DBMS.

NGINX redirects client connections to pre-forked PHP processes using *FastCGI* over sockets, applying a well-understood balance between isolating requests across separate PHP processes, and avoiding the cost of per-request PHP process creation. We use PostgreSQL because it is fully-featured (vs. SQLite) and yet has a small memory – and thus EPC – footprint (e.g., compared to MySQL). PostgreSQL spawns at least six processes that perform different functions; each process is derived from the same binary, and all processes use shared memory to exchange data (see Fig. 8).

Fig. 7 shows two deployments for our web service: the first uses vanilla SGX-LKL [58] to deploy the application in multiple SGX TEEs; the second uses a single SGX TEE with SSF. Note that SSF can isolate the web server and its SSL library, which is not possible in the multi-TEE configuration without compromising performance. Existing solutions also cannot isolate shared libraries, and previous approaches to emulating isolated processes within an SGX TEE [72] lack the shared memory support required to run PostgreSQL.

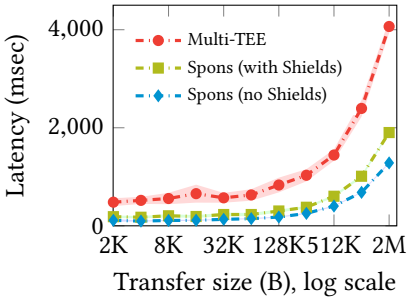


Figure 10. Request latency (Multi-process web service)

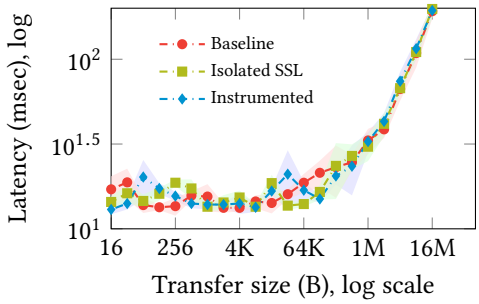


Figure 11. Request latency (Single-process NGINX)

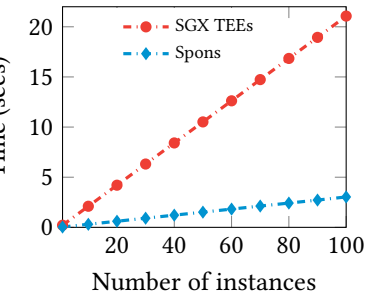


Figure 12. Creation time (Spons vs. SGX TEE)

Fig. 7b shows how we map the application to SSF. Spon and Shield pairs transparently replace processes, and `execve` replaces `fork` + `exec`. NGINX isolates its OpenSSL library and the private encryption keys into a separate passive Spon with a nested Shield, and all PostgreSQL Spons use a single Shield to allow shared memory (see Fig. 8). We compile all components using SSFPass, link them against `spn-musl`, change process creation to use `execve`, and configure the loader to create appropriate Spons and Shields. We also minimally modify NGINX to isolate its OpenSSL component into a nested Shield using `spon_call` (the OpenSSL function wrappers do not require data marshalling and can directly operate on the pointers provided by NGINX).

Python cryptographic library. PyCryptodome [54] is a popular cryptography framework for Python, implemented as a wrapper over C functions. High-level cryptographic operations in Python invoke binary modules written in C that can e.g., use hardware cryptographic instructions such as AES-NI. Since the low-level cryptographic modules may have memory safety issues, we use Shields to isolate them.

Fig. 9 shows the design of Python code that isolates each of the PyCryptodome modules (i.e., ciphers written in C) using separate passive Spons. A single Python interpreter uses a set of Spon-Shield pairs. PyCryptodome's indirection layer manages the modules, creating the necessary Spons and Shields and invoking their functions. We create a new Spon-Shield pair for each encryption context in PyCryptodome, which contains functions that can be called multiple times.

The web service and PyCryptodome applications showcase the use of asymmetric memory protection in opposite ways. In the web service, NGINX calls into a protected OpenSSL encryption library, whereas in PyCryptodome, the Python interpreter protects itself from the memory-unsafe cryptographic functions that it invokes. As a consequence, PyCryptodome's indirection layer is responsible for (un)marshalling the arguments and results for the C-module functions into/from their assigned Shield (via `alloc_mem`; see §3.3).

6.2 Multi-Process Web Service

We deploy the multi-process web service with SSF with a PHP-based *content management system* (CMS) that serves markdown pages stored in the DBMS. We generate 500 MB of markdown files of 2 KB to 2 MB in size, and measure the request latency of a client requesting random pages.

Fig. 10 compares the request latency for different file sizes with SSF to a multi-TEE deployment. (The data points show the mean over 10 runs for each file size; the shaded extent indicates one standard deviation.) The distribution of latencies is always similar, but SSF provides a consistent improvement. Small file sizes (2–32 KB) have an almost constant latency that grows linearly with the file size, because PostgreSQL creates a new Spon on each connection, which requires constant time (tied to the size of the Spon); for larger files, the overhead is small in comparison to the data processing and transfer costs across components. The deployment with Spons and Shields has, on average, 2.6× lower request latencies than the multi-TEE one; a deployment without Shields achieves 4.4× lower latencies. (CPU load is consistent.)

These results confirm that, in the multi-TEE deployment, the need to use encryption between TEEs and the absence of caches adds more overhead than the SSF's instrumentation.

6.3 Python Cryptographic Library

To measure the overhead when passive Spons are invoked, we compare the performance of PyCryptodome when executed inside a TEE in three configurations: (i) without Spons as a baseline; (ii) with Spons but without memory instrumentation; and (iii) with Spons and memory instrumentation.

As a workload, we use the `pct-speedtest.py` benchmark provided by PyCryptodome. It measures the performance of various cryptographic operations such as encryption and key initialisation for ciphers. We focus on one benchmark, AESNI, with two forms of the AES cipher (GCM and CTR) with different key lengths (128, 192 and 256 bits).

Fig. 13 shows the key set-up speed, measured in thousands of key initialisations per second. As can be seen from the results, while different ciphers have different performance characteristics, there is a similar trend across all experiments: the use of Spons reduces performance by 1%–3%. Memory instrumentation adds a further 1%–2% overhead. This is smaller than in the previous experiment with active Spons because this workload dereferences fewer pointers.

Next we consider the encryption performance. As Fig. 14 shows, the impact of Spons here is more significant. Using

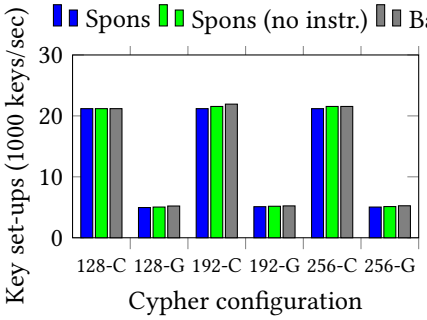


Figure 13. Key set-up benchmark

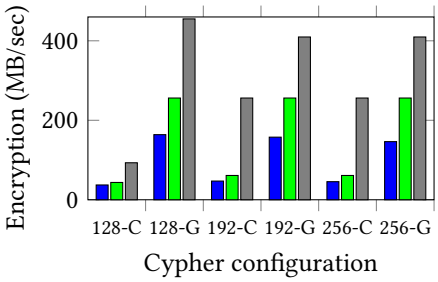


Figure 14. AES-NI benchmark

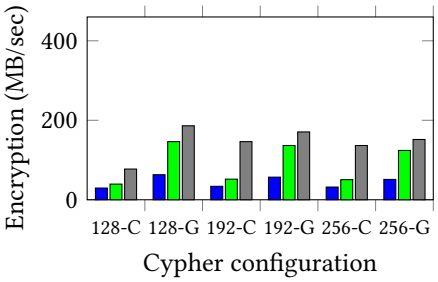


Figure 15. AES benchmark

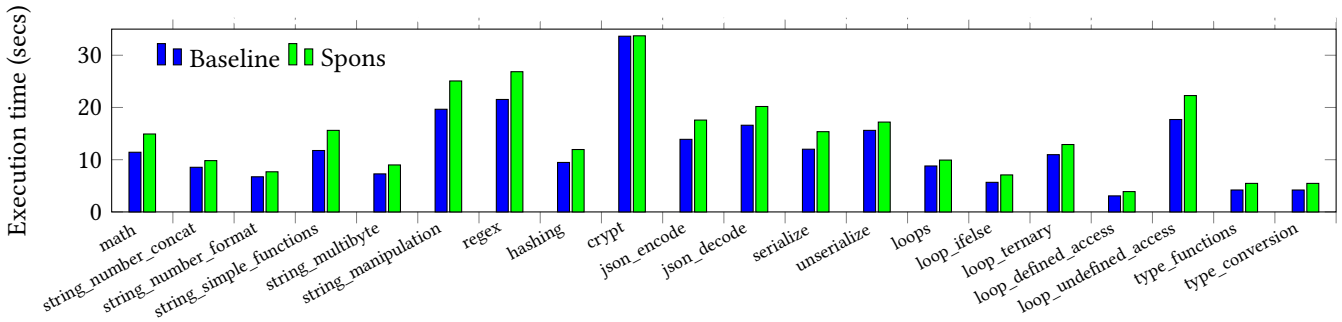


Figure 16. Performance of various PHP functions for instrumented and non-instrumented php-fcgi compartments (The instrumentation decreases performance by 22% on average.)

Spons decreases performance by 53%–76% for CTR, and by 38%–44% for GCM. Adding memory instrumentation reduces performance further by 5%–7% for CTR, and by 2% for GCM.
.

In the previous experiment, the time spent on memory copies is significant compared to computation time. If computation is expensive, the overhead decreases: the results in Fig. 15 are from the same experiment but without hardware-accelerated AES-NI instructions. The impact of using Spons is also lower: on average, non-instrumented Spons have 57% lower performance for CTR, and 21% for GCM.

6.4 Instrumentation Overhead

Next we benchmark Spon instances to determine if there is a difference in overhead for different types of computation inside Spons. We also consider the impact on binary size.

Since bounds-checking overheads depend on the specific instrumented code, we explore the incurred overheads using three workloads: (1) we run a PHP interpreter inside a Spon and use a benchmark suite [66] that measures the performance of various PHP functions. In contrast to the experiments in §6.2, we directly invoke the PHP interpreter to remove the data exchange overhead between the web server and the PHP worker; (2) we use the pgbench benchmark from PostgreSQL, providing TPC-B-like queries with five `SELECT`, `UPDATE`, and `INSERT` commands per transaction [56, 57]. We use a scale factor of 10 to generate the initial database, which results in 1,000,000 rows, or almost

Table 3. Impact of SSF's memory instrumentation

	PHP		PostgreSQL		NGINX
	Size	Time	Size	Tx per sec.	Size
No instr.	16 MB	243 s	11 MB	528	6.3 MB
Instr.	27 MB	292 s	21 MB	473	11 MB

330 MB of data; and (3) we use the NGNIX web server with the OpenSSL library, and have clients fetching static objects of different sizes.

The PHP benchmark suite allows us to compare the performance of PHP functions; pgbench and NGINX benchmarks, measure overall performance. The NGINX benchmark helps us understand the cost of an asymmetric Spon with OpenSSL.

Performance overhead. For all three benchmarks, we compare deployments under two configurations: (i) an uninstrumented baseline and (ii) Spons with instrumentation.

Tab. 3 shows that the instrumented version has an increase in total execution time of 20%. Fig. 16 gives a detailed breakdown of the PHP results: the average overhead is 22% across all benchmarks, with performance degrading from 1.55 to 1.26 million operations per second.

The table also shows the results of the PostgreSQL benchmark. Compared to PHP, PostgreSQL exhibits only a 10% performance decrease: the number of transactions changes from 528 to 473 transactions per second.

In contrast to the Python benchmark, NGINX does not copy data to/from the Spon, and thus the invocation should have less overhead. To measure this, we generate 513 MB

of random files and fetch them via HTTPS using the `curl` utility in a single thread. In total, we have 30 files (with sizes from 1 byte to 256 MB, each size a power of 2). We fetch each remote file at least 10 times and measure request latency.

Fig. 11 shows the results for (i) a baseline without instrumentation; (ii) a Spon with instrumentation only; and (iii) a Spon with instrumentation and isolated OpenSSL. For sizes smaller than 100 KB, the response time is the same, and there is no significant overhead from the instrumentation or the use of Spons. For bigger sizes, the response time grows linearly with size, which shows that the overhead is negligible compared to the traffic encryption time.

Binary size overhead. SSFPass's (see §4.2) memory instrumentation adds new instructions each time the target code accesses pointers. The stripped binary of `php-fcgi` is 16 MB, whereas the Spon-based PHP interpreter is 1.7× larger at 27 MB (see Tab. 3). The instrumentation increases the size of the NGINX binary, which also includes the statically-linked OpenSSL library, by 1.7× (11 MB vs. 6.3 MB), and the size of the PostgreSQL binary by 1.9× (21 MB vs. 11 MB).

6.5 Instantiation Time

Finally, we compare the time to instantiate Spons against SGX TEEs using the SGX-LKL TEE OS. Our simple benchmark measures the time to create sequentially 100 Spons and SGX TEEs, respectively, in batches of 10. We deploy a simple program in each enclave, and limit the enclave size to 8 MB.

Fig. 12 shows that both SSF and regular SGX TEEs scale linearly in the number of instances, but Spons require significantly less time. On average, Spons are created in 30 ms, while the regular SGX TEE needs 210 ms.

7 Related Work

Partitioning frameworks. Wedge [7] creates isolation entities inside processes using a *default-deny* security model. Wedge's *Crowbar* tool helps developers find which parts of the program to isolate. PrivTrans [10] is a source-level partitioning tool that splits an application into two separate parts, trusted and untrusted. Both PrivTrans and Wedge rely on the kernel for isolation and cannot be used within some TEEs.

Glamdring [40] partitions an application to use SGX TEEs according to source-level annotations. SOAAP [22] is an LLVM-based tool to help developers reason about what to isolate. Such partitioning policies could potentially be enforced using SSF's isolation primitives.

Panoply [73] is an SGX-based partitioning infrastructure that supports the `fork` system call by spawning a new TEE and copying data from the parent enclave: stronger isolation but with higher overhead. GOTEE [19] enables automatic partitioning of applications written in Go into TEE and untrusted code, but does not isolate application components within a TEE. Komodo [17] proposes a flexible software-defined TEE model. SSF provides interfaces for compartmentalising applications inside TEEs, that could use Komodo.

Coarse-grained sandboxing. Past work on *software fault isolation* (SFI) [20, 69, 81] shares SSF's goal of coarse-grained sandboxing. Like MemSentry [36], SSFPass uses Intel MPX instead of software-only sandboxing to improve performance, but SSF's use of MPX differs—MemSentry adds one bound check but relies on Intel MPK and Intel's VM functions (VMFUNC). SSFPass adds two bounds checks but is not restricted in the number of domains (MPK) or dependent on technology unavailable inside some TEEs (VMFUNC). SEIMI [82] offers an intra-process isolation technique based on Supervisor-mode Access Prevention (SMAP), but requires code to be executed in privileged mode. Janus [24, 25] supports switching of protection domains without involving the kernel, but also relies on either MPK or VMFUNC. As well as sandboxing mechanisms, SSF provides a flexible abstraction for isolation of existing applications, and addresses TEE-specific issues such as secure and efficient interaction with a TEE OS.

Moat [75] and SIR [74] separate TEE code into untrusted and trusted parts, and certify that untrusted machine code cannot leak confidential information. However, similar to ConfLLVM [8], they only support two compartments. MPTEE [88] uses Intel MPX to provide memory isolation of program regions and implements trusted memory attributes missing in Intel SGXv1. This technology can be potentially applied in SSF to enforce its control-flow integrity.

Embedded systems. Since embedded systems often lack MMU support, recent work has used memory-protection units (MPUs) for compartmentalisation (e.g., for ARM [15, 33] and RISC-V [85]). Similar to Intel MPK, MPUs are restricted in the maximum number of domains (16 on some ARM hardware). TIMBER-V [85] addresses this limitation by combining MPUs with tagged memory, but requires custom hardware support unavailable in SGX TEEs. None of these approaches allow for secure and efficient TEE OS interaction.

8 Conclusions

TEEs isolate applications from the host system software and even physical attacks but vulnerabilities in TEE code remain an issue. Fine-grained compartmentalisation increases security through defense-in-depth, but current solutions sacrifice performance and compatibility. We introduce *Spons* and *Shields*, new flexible isolation abstractions for TEEs. Spons are self-contained isolated memory regions that can behave like sandboxed libraries or processes. Spons are isolated inside Shields by the compiler and use hardware acceleration for bounds-checking. We show how Spons and Shields can help compartmentalise an existing application inside a TEE and to port a multi-process application to a TEE.

Acknowledgements

This work was partially funded by the UK Government's Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. 340–353.

[2] Alibaba Cloud. 2020. ECS Bare Metal Instance. https://www.alibabacloud.com/product/ebm. Last accessed: March 8, 2021.

[3] AMD 2020. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More.* AMD.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 689–703.

[5] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A Fork() in the Road. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM, 14–22.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3, Article 8 (Aug. 2015), 26 pages.

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. USENIX Association.

[8] Ajay Brahmakshatriya, Piyus Kedia, Derrick P. McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. 2019. ConfLLVM: A Compiler for Enforcing Data Confidentiality in Low-Level Code. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, Article 4, 15 pages.

[9] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. ACM, Article 14, 13 pages.

[10] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, San Diego, CA.

[11] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23*. 985–999.

[12] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 645–658.

[13] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 253–264.

[14] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 56–71.

[15] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. ACES: Automatic Compartments for Embedded Systems. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 65–82.

[16] Intel Corp. 2014. Software Guard Extensions Programming Reference, Ref. 329298-002US. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[17] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, 287–305.

[18] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX: In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 83–97.

[19] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 571–586.

[20] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. ACM.

[21] Graphene-SGX Source Code 2021. Graphene Library OS with Intel SGX Support. https://github.com/oscarlab/graphene. Last accessed: March 8, 2021.

[22] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAPP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1016–1031.

[23] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Commun. ACM* 52, 5 (May 2009), 91–98.

[24] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael Scott, Kai Shen, and Mike Marty. 2018. *Janus: Intra-process isolation for high-throughput data plane libraries.* Technical Report. Technical Report UR CSD/1004.

[25] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX ATC*. USENIX Association, Renton, WA, 489–504.

[26] ARM Holdings. 2009. ARM Security Technology: Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/.

[27] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 393–405.

[28] IBM Cloud. 2020. IBM Cloud Data Shield. https://www.ibm.com/cloud/data-shield. Last accessed: March 8, 2021.

[29] Intel. 2018. Intel® 64 and IA-32 Architectures Software Developer's Manual. (2018).

[30] Intel. 2020. White paper: Intel Trust Domain Extensions. https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf.

[31] Simon P. Johnson. 2019. Scaling Towards Confidential Computing. https://systex.ibr.cs.tu-bs.de/systex19/slides/systex19-keynote-simon.pdf.

[32] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[33] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *NDSS*.

[34] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 61–72.

[35] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19.

[36] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on

Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, 437–452.

[37] S. Kuenzer, S. Santhanam, Y. Volchkov, F. Schmidt, F. Huici, Joel Nider, Mike Rapoport, and Costin Lupu. 2019. Unleashing the Power of Unikernels with Unikraft. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19)*. ACM, 195.

[38] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, 205–221.

[39] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75–86.

[40] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 285–298.

[41] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 973–990.

[42] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 49–64.

[43] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, 1607–1619.

[44] LLVM. Last accessed: March 8, 2021. Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[45] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 461–472.

[46] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 1–9.

[47] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. *HASP@ ISCA* 10 (2013).

[48] Microsoft Azure. 2021. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute. Last accessed: March 8, 2021.

[49] Paul Muntean, Mathias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2020. ρFEM: Efficient Backward-Edge Protection Using Reversed Forward-Edge Mappings. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. ACM, 466âĂŞ479.

[50] musl libc. 2021. https://www.musl-libc.org. Last accessed: March 8, 2021.

[51] nginx, an HTTP server. 2021. https://www.nginx.org. Last accessed: March 8, 2021.

[52] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (2018), 28.

[53] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*. ACM, 238–253.

[54] Panoply 2021. A self-contained cryptographic library for Python. https://github.com/Legrandin/pycryptodome. Last accessed: March 8, 2021.

[55] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 241–254.

[56] PostgreSQL. 2021. A Simple Benchmark Program for PostgreSQL. https://github.com/postgres/postgres/tree/master/src/bin/pgbench. Last accessed: March 8, 2021.

[57] PostgreSQL. 2021. PostgreSQL 12.3 Documentation. https://www.postgresql.org/docs/12/index.html. Last accessed: March 8, 2021.

[58] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *arXiv preprint arXiv:1908.11143* (2019).

[59] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *12th USENIX Security Symposium (USENIX Security 03)*. USENIX Association.

[60] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. 2010. LKL: The Linux Kernel Library. In *9th RoEduNet IEEE International Conference*. IEEE, 328–333.

[61] Ramu Ramakesavan, Dan Zimmerman, and Pavithra Singaravelu. 2015. Intel Memory Protection Extensions (Intel MPX) Enabling Guide.

[62] Charlie Reis. 2018. Mitigating Spectre with Site Isolation in Chrome.

[63] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1661–1678.

[64] Lars Richter, Johannes Götzfried, and Tilo Müller. 2016. Isolating Operating System Components with Intel SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*. 1–6.

[65] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security* 15, 1, Article 2 (March 2012), 34 pages.

[66] RuSoft. 2021. PHP benchmark script. https://github.com/rusoft/php-simple-benchmark-script. Last accessed: March 8, 2021.

[67] Vasily Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, 575–587.

[68] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. 2018. *E*Actors: Fast and Flexible Trusted Computing Using SGX. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. ACM, New York, NY, USA, 187–200.

[69] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium (USENIX Security 10)*. USENIX Association.

[70] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, 552–561.

[71] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing

Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19)*. ACM, Article 8, 11 pages.

[72] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *ASPLOS '20*.

[73] Shweta Shinde, DL Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*. 12.

[74] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. 2016. A Design and Verification Methodology for Secure Isolated Regions. In *PLDI '16*. ACM, 17 pages.

[75] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2015. Moat: Verifying Confidentiality of Enclave Programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, 1169–1184.

[76] The Apache Software Foundation. The Apache HTTP server. 2021. https://www.apache.org. Last accessed: March 8, 2021.

[77] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *USENIX Security Symposium*. USENIX Sec.

[78] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1221–1238.

[79] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 1041–1056.

[80] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting Software with Code-centric Memory Domains. In *Intl. Symp. on Computer Architecture (ISCA)*. 469–480.

[81] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. *SIGOPS Oper. Syst. Rev.* 27, 5 (Dec. 1993), 203–216.

[82] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang. 2020. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 592–607.

[83] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.

[84] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*. Springer, 440–457.

[85] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *NDSS*.

[86] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE, 640–656.

[87] Bin Zeng, Gang Tan, and Greg Morrisett. 2011. Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, 29–40.

[88] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. 2020. MPTEE: Bringing Flexible and Efficient Memory Protection to Intel SGX. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. ACM, Article 18, 15 pages.